Hybrid Dragonfly Algorithm for the Traveling Salesman Problem

This code implements a Hybrid Dragonfly Algorithm for the Traveling Salesman Problem (TSP), which is a classic optimization problem in computer science. The goal is to find the shortest possible route that visits a given set of cities exactly once and returns to the starting city.

The algorithm consists of the following steps:

1. Generate city coordinates: The `generate_city_coordinates` function generates random (x, y) coordinates for a given number of cities.

2. Generate distance matrix: The `generate_distance_matrix` function calculates the Euclidean distance between each pair of cities using their coordinates and returns a matrix containing the distances.

3. Calculate total distance: The `calculate_total_distance` function takes a tour (a sequence of cities) and the distance matrix as input, and calculates the total distance of the tour.

4. 2-opt local search: The `two_opt_local_search` function applies the 2-opt heuristic to improve a given tour. The 2-opt heuristic is a local search algorithm that iteratively swaps two edges in the tour to reduce its length.

5. Dragonfly movement: The `dragonfly_movement` function performs a random swap of two cities in a given tour and accepts the new tour if its cost is lower than the original tour. This is a strategy for exploring new solutions in the search space.

6. Hybrid Dragonfly Algorithm: The `hybrid_dragonfly_algorithm` function combines the above steps to find the best tour using a population of dragonflies. It initializes a population of dragonflies with random tours, performs local search and dragonfly movement iteratively for a fixed number of iterations, and updates the best tour if a better solution is found.

7. Plot tour: The `plot_tour` function plots the best tour found by the algorithm using Matplotlib.

Code Explanation:

```
import random
import numpy as np
import matplotlib.pyplot as plt
```

This imports the required libraries for the algorithm: `random` for generating random numbers, `numpy` for numerical computations, and `matplotlib` for plotting the results.

```
def generate_city_coordinates(num_cities):
    coordinates = []
    for _ in range(num_cities):
        x, y = random.uniform(0, 100), random.uniform(0, 100)
        coordinates.append((x, y))
    return coordinates
```

This function generates random (x, y) coordinates for a given number of cities between 0 and 100.

```
def generate_distance_matrix(coordinates):
    num_cities = len(coordinates)
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            distance = np.sqrt((coordinates[i][0] - coordinates[j][0]) ** 2
+ (coordinates[i][1] - coordinates[j][1]) ** 2)
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance
    return distance_matrix
```

This function calculates the Euclidean distance between each pair of cities using their coordinates and returns a matrix containing the distances.

```
def calculate_total_distance(tour, distance_matrix):
    total_distance = 0
    num_cities = len(tour)
    for i in range(num_cities - 1):
        current_city = tour[i]
        next_city = tour[i + 1]
        total_distance += distance_matrix[current_city][next_city]
    total_distance += distance_matrix[tour[-1]][tour[0]]
    return total_distance
```

This function takes a tour (a sequence of cities) and the distance matrix as input, and calculates the total distance of the tour by summing up the distances between consecutive cities and adding the distance from the last city to the starting city.

```
def two_opt_local_search(tour, distance_matrix):
    num_cities = len(tour)
    best_tour = tour.copy()
    best_cost = calculate_total_distance(tour, distance_matrix)

    improved = True
    while improved:
        improved = False
        for i in range(1, num_cities - 1):
            for j in range(i + 1, num_cities):
                if j - i == 1:
                    continue
                new_tour = best_tour[:i] + best_tour[i:j][::-1] +
best_tour[j:]
                new_cost = calculate_total_distance(new_tour,
distance_matrix)
                if new_cost < best_cost:
                    best_cost = new_cost
                    best_tour = new_tour
                    improved = True
    return best_tour
```

This function applies the 2-opt heuristic to improve a given tour. The 2-opt heuristic is a local search algorithm that iteratively swaps two edges in the tour to reduce its length. The function keeps swapping edges until no further improvement can be made.

```
def defdragonfly_movement(tour, best_tour, distance_matrix):
    num_cities = len(tour)
    new_tour = tour.copy()
    for _ in range(2):  # Try two random swaps
        i, j = random.sample(range(num_cities), 2)
        new_tour[i], new_tour[j] = new_tour[j], new_tour[i]
    if calculate_total_distance(new_tour, distance_matrix) <
calculate_total_distance(tour, distance_matrix):
        return new_tour  # Accept the new tour if its cost is lower
    return tour  # Otherwise, return the original tour
```

This function performs a random swap of two cities in a given tour and accepts the new tour if its cost is lower than the original tour. This is a strategy for exploring new solutions in the search space.

```
def hybrid_dragonfly_algorithm(num_dragonflies, num_cities,
num_iterations):
    # Generate the city coordinates
    coordinates = generate_city_coordinates(num_cities)
    # Generate the distance matrix
    distance_matrix = generate_distance_matrix(coordinates)
    # Initialize the dragonflies population
    population = []
    for _ in range(num_dragonflies):
        tour = random.sample(range(num_cities), num_cities)
        population.append(tour)
    # Initialize the best tour and its cost
    best_tour = population[0]
    best_cost = calculate_total_distance(best_tour, distance_matrix)
    # Perform iterations
    for iteration in range(num_iterations):
        # Perform local search on a subset of dragonflies
        for i in range(num_dragonflies):
            population[i] = two_opt_local_search(population[i],
distance_matrix)
        # Perform dragonfly movements
        for i in range(num_dragonflies):
            population[i] = dragonfly_movement(population[i], best_tour,
distance_matrix)
        # Update the best tour if a better solution is found
        for tour in population:
            cost = calculate_total_distance(tour, distance_matrix)
            if cost < best_cost:
                best_tour = tour
                best_cost = cost
    return best_tour, best_cost
```

This function combines the above steps to find the best tour using a population of dragonflies. It initializes a population of dragonflies with random tours, performs local search and dragonfly movement iteratively for a fixed number of iterations, and updates the best tour if a better solution is found.

```
def plot_tour(coordinates, best_tour):
    x = [coordinates[i][0] for i in best_tour] +
[coordinates[best_tour[0]][0]]
    y = [coordinates[i][1] for i in best_tour] +
[coordinates[best_tour[0]][1]]
```

```
    plt.figure(figsize=(10, 6))
    plt.plot(x, y, '-o', color='blue', linewidth=1, markersize=5)
    plt.ylabel('Y-Coordinates')
    plt.xlabel('X-Coordinates')
    plt.title('Best TSP Tour')
    plt.grid()
    plt.show()
```

This function plots the best tour found by the algorithm using Matplotlib. It takes the city coordinates and the best tour as input and plots the tour as a line connecting the cities.

```
# Example
num_cities = 10
# Generate city coordinates
coordinates = generate_city_coordinates(num_cities)
# Generate distance matrix using coordinates
distance_matrix = generate_distance_matrix(coordinates)
num_iterations = 1000
best_tour, best_cost = hybrid_dragonfly_algorithm(100, num_cities,
num_iterations)
print("Best tour:", best_tour)
print("Cost:", best_cost)
# Plot the best tour
# plot_tour(coordinates, best_tour)
```

This is an example usage of the algorithm. It generates random city coordinates, calculates the distance matrix, and runs the hybrid dragonfly algorithm for a fixed number of iterations. It prints the best tour and its cost and optionally plots the tour using the `plot_tour` function.